Real-time GC

Review

- Parallel has expected better throughput
- Concurrent has expected better latency
- Concurrent GC sacrifices mutator time to assure collector progresses
- Sacrificed time may have unknown bounds

Real-time systems

- Soft real-time: Actions have time limits. If time limits are not met, system has, e.g., degraded quality or performance
- Hard real-time: Actions have time limits. If time limits are not met, the system fails.
- Safety-critical: Actions have time limits. If time limits are not met, people die.

Real-time systems

- Soft real-time: Actions have time limits. If time limits are not met, system has, e.g., degraded quality or performance
- Hard real-time: Actions have time limits. If time limits are not met, the system fails.
- Safety-critical: Actions have time limits. If time limits are not met, people die.

Most real-time GCs live here

Real-time systems

- Soft real-time: Actions have time limits. If time limits are not met, system has, e.g., degraded quality or performance
- Hard real-time: Actions have time limits. If time limits are not met, the system fails.
- Safety-critical: Actions have time limits. If time limits are not met, people die.

Most real-time GCs live here

No GCs live here, but some try

Real-time systems

- Soft real-time: Actions have time limits. If time limits are missed, system has, e.g., degraded quality or performance
- Hard real-time: Actions have time limits. If time limits are missed, the system fails.
- Safety-critical: Actions have time limits. If time limits are missed, people die.

Most real-time GCs live here

No GCs live here, but some try

Here there be dragons

Examples

- Soft real-time: Video playback, games, communication systems, aviation/spaceflight guidance
- Hard real-time: Mechanical control systems, robotics
- Safety-critical: Aviation/spaceflight control and life support, self-driving cars, medical devices

Real-time systems

- To achieve hard real-time, must worry about:
  - Compiler (e.g. optimizations)
  - Runtime system (e.g. GC)
  - Operating system (e.g. scheduler)
  - Architecture (e.g. caching, pipelining, NUMA)

Real real-time systems

- Throughput almost doesn't matter
- Common solution is to sacrifice throughput for predictability:
  - Compiler: Turn off all optimizations, avoid virtual dispatch, avoid nondeterminism
  - Runtime systems: Don't allocate
  - Operating systems are for the weak
  - Architecture: Cache-free memory access, simple uniprocessor

Real-time analysis

- What we care about is worst-case execution time
- Sacrificing best-case execution for worst-case execution is fine
- Worst-case execution cannot average in any sense: Any action could happen at the worst time

A fool’s errand?

- With all these restrictions, real-time GC may seem impossible
  (… and perhaps it is)
- However, predictability implies *no bugs*
- There is a desire for real-time GC to avoid error-prone manual memory management

Cheating

- Total time to GC is fixed by heap size
- So here's a hard real-time system: For every mutator operation, run GC
- In practice, throughput does still matter

When to GC

- This is the principal difference in real-time GC: Must GC when harmless
- Work-based: For every *n* seconds of mutator work, GC gets *tn* seconds
- Slack-based: Run when mutator is idle
- Time-based: GC gets time on a regular interval, regardless of mutator activity

When to GC

- This is the principal difference in real-time GC: Must GC when harmless
- Work-based: For every *n* seconds of mutator work, GC gets *tn* seconds

"Tax rate"

- Slack-based: Run when mutator is idle
- Time-based: GC gets time on a regular interval, regardless of mutator activity

Baker incremental semispace

- Work-based: Mutator sacrifices predictable time
- Single-threaded, incremental: Mutator pauses for fixed time to perform incremental work
- Fixed-size: All objects are same size (Lisp `cons` cells)

Baker mutator barrier

Baker barrier:

```
read(obj, loc):
   if obj in fromspace:
      ref := handle(*(obj+loc))
   else:
      ref := *(obj+loc)
   return ref
```

Baker mutator barrier

Baker barrier:

```
read(obj, loc):
   if obj in fromspace:
      ref := handle(*(obj+loc))
   else:
      ref := *(obj+loc)
   return ref
```

Fixed-size object,
fixed-time handling

Baker: When to GC

- Work-based: Mutator pays based on work
- Must do enough GC that allocation will always succeed
- So, e.g., if cycle starts with heap half-empty, must scan at least one object per allocation
- Note: Cons cells have only one reference, so can at most copy one object: One-per-one

Problem

- Imagine: 50% of heap is full, so start GC…
- 25% is kept, mutator allocates 50% during collection
- Heap is now 75% full, so start GC…
- Mutator allocates 25%, GC not finished, stall!

More problems

- With collection work during allocation, L can only grow
- Must tax other operations for L to ever shrink (typically write barrier)
- There is no perfect solution: Pathologies are unavoidable. Thus, soft real-time.

Blelloch and Cheng replicating

- Problem with multiple-sized objects is unpredictable time to scan and (as applicable) handle children
- Must make predictable:
  - Time to copy object
  - Time to scan object

Blelloch and Cheng replicating

- Basic idea: Don’t copy/scan whole objects, copy/scan words at a time
- Kept in sync by *replication*: Mutator must update both fromspace and tospace copies
- Instead of waiting for whole objects to copy, wait only for words

Cheng headers

```
struct Header {
   struct GCTypeInfo *type;
   void *forwardingAddress;
   size_t copyingWord;
};
```

```
write(obj, loc, val):
  if isPointer(obj, loc):
    assureCopied(obj[loc])
  obj[loc] := val

  if obj->forwardingAddress:
    while obj->forwardingAddress == PLEASE_HOLD: wait
    to := obj->forwardingAddress

    while obj->copyingWord == loc: wait

    if isPointer(obj, loc):
      val := assureCopied(val)
    to[loc] := val

  collectSomeWords()

read(obj, loc):
  return obj[loc]
```

```
write(obj, loc, val):
  if isPointer(obj, loc):
    assureCopied(obj[loc])
  obj[loc] := val

  if obj->forwardingAddress:
    while obj->forwardingAddress == PLEASE_HOLD: wait
    to := obj->forwardingAddress

    while obj->copyingWord == loc: wait

    if isPointer(obj, loc):
      val := assureCopied(val)
    to[loc] := val

  collectSomeWords()

read(obj, loc):
  return obj[loc]
```

"Please hold" means "This object is being examined, but tospace has not yet been allocated."

```
write(obj, loc, val):
  if isPointer(obj, loc):
    assureCopied(obj[loc])
  obj[loc] := val

  if obj->forwardingAddress:
    while obj->forwardingAddress == PLEASE_HOLD: wait
    to := obj->forwardingAddress

    while obj->copyingWord == loc: wait

    if isPointer(obj, loc):
      val := assureCopied(val)
    to[loc] := val

  collectSomeWords()

read(obj, loc):
  return obj[loc]
```

"Please hold" means "This object is being examined, but tospace has not yet been allocated."

Maximum pause is duration of one word-copy

```
assureCopied(obj):
  if testAndSet(&obj->forwardingAddress,
                NULL, PLEASE_HOLD):
    size := obj->type->size
    to := atomicIncrement(&tospace->free, size)
    to->copyingWord := size
    from->copyingWord := 0
    from->forwardingAddress := to
    addToWorklist(from)
  else:
    while obj->forwardingAddress == PLEASE_HOLD: wait
  return obj->forwardingAddress
```

Collection

- We can collect *k* words by keeping track of `copyingWord`
- Collector updates tospace references to always point at tospace, as does write barrier
- Same pathologies as Baker

Post-collection

- fromspace points at fromspace,
  tospace points at tospace,
  roots point at fromspace
- Simply update roots to point at tospace and we’re done
- … “simply” ain’t simple, another real-time problem

Work-based collection

- These algorithms have been work-based
- Barrier during collection is very slow: A single write operation could be dozens of synchronized operations
- Must assume worst case
- Worst-case time is simply too bad for many uses

Henriksson slack-based

- Run GC during slack time, and
- never sacrifice (much) mutator time of “high-priority” tasks
- Since tasks have guaranteed time bounds, we hopefully can guarantee enough slack time

Henriksson

- Basically a conventional concurrent copying GC
- Increments carefully timed to assure they collect at least as much as can be allocated
- Do not tax mutators: If mutator touches already-copied object, copy invalidated

Henriksson problems

- Mutator can effectively undo collection work
- Pathology: Mutator repeatedly updates fromspace objects, invalidating tospace copies, collector cannot terminate
- Requires very precise knowledge of task scheduling to guarantee enough time is given

Metronome time-based

- Divide execution time into mutator *quanta* and collector quanta
- Collector quanta occur at fixed intervals, for an exact time pre-specified (typically 30% of execution time)
- Up to programmer to assure that's enough
- Otherwise similar to work-based

Best of both worlds?

- Auerbach tax-and-spend systems:
  - Mutator is normally taxed
  - If thread has slack time, collection in that time builds up *credits*
  - Credits can be spent to avoid taxation
- Allows slack-based collection without forethought about balance

Course conclusions

The world of GC

- GC is a world of tradeoffs
- Some tradeoffs can be made on the fly, others only by choosing ahead of time
- The best GC depends on:
  - Expected lifetime of objects, size of objects, size variability of objects, available heap, performance concerns, throughput vs latency, processor availability, language tractability, language features…

So what's best?

- Opinions corner:
  - Common throughput-bound applications: Classic generational GC ala Java
  - Common latency-bound applications: Concurrent generational GC
  - Systems with unusual object properties: Mark-and-sweep
  - Real-time systems: No GC (maybe ownership?)

What now

- Concurrent and real-time GC are areas of very active research
- If you haven’t, play with Java’s several GCs
- Remember: presentations next week, final projects due on final week

Final presentations

- Please bring a USB stick with your presentation
- If you would like, bring a laptop to present from (USB stick for backup)
- If you'd strongly prefer, talk to me about whiteboard-only presentation